Mini OS - 파일탐색기 제작

20202990 이정우

20221598 박찬우

20180610 전준영

목차

1.서론

1-1. 프로젝트 배경

우분투를 포함한 많은 리눅스 배포 판 사용자들은 터미널을 통해 시스템을 관리하고 파일을 조작하는 경우가 많습니다. 그러나 터미널 기반 파일 탐색기는 사용자가 명령어를 직접 입력해야 하므로 불편함이 따릅니다. 이에 따라, 터미널 환경에서도 GUI 파일 탐색기처럼 직관적으로 파일을 탐색할 수 있는 방법을 모색하게 되었습니다.

1-2. 프로젝트 목적

본 프로젝트의 목적은 배포된 miniOS를 이용하여 우분투 터미널에서 GUI 파일 탐색기와 유사한 기능을 제공함으로써, 터미널 사용자들의 파일 관리 작업을 보다 효율적이고 편리하게 만드는 것입니다.

1-3. 역할 분담

박찬우 – 터미널 기반 파일 탐색 및 실행 시스템 구현

전준영 – 터미널 파일 탐색 및 정렬 시스템

이정우 – 터미널 파일 및 폴더 관리 시스템

2. 방법

2-1. 환경 설정

i. 개발환경 설정

이번 프로젝트는 Ubuntu 환경에서 개발되었습니다. Ubuntu는 안정적이고 널리 사용되는 리눅스 배포판으로, 다양한 개발 도구와 라이브러리를 지원하여 개발 및 테스트에 적합합니다. 특히 멀티스레딩 및 파일 시스템 관련 작업에 강력한 성능을 발휘하여 파일 탐색기의 구현 및 성능 최적화에 유리한 환경을 제공합니다.

ii. 헤더파일 설정

```c
#define MAX_THREADS 3 // 최대 스레드 수

static int running_threads = 0; // 현재 실행 중인 스레드 수
static pthread_mutex_t running_threads_mutex = PTHREAD_MUTEX_INITIALIZER;

void scheduleTask(void *(*task)(void *), void *arg) {
    pthread_mutex_lock(&running_threads_mutex);
    if (running_threads < MAX_THREADS) {
        running_threads++;
        pthread_mutex_unlock(&running_threads_mutex);

        pthread_t thread;
        pthread_create(&thread, NULL, task, arg);
        pthread_detach(thread);
    } else {
        pthread_mutex_unlock(&running_threads_mutex);
        usleep(1000000); // 0.1초 대기
        printf("thread is waiting..\n");
        scheduleTask(task, arg); // 재귀적으로 호출하여 다시 시도
    }
}
```

'scheduleTask' 함수는 주어진 태스크를 스레드 풀의 제한에 따라 스케줄링한다. 만약 현재 실행 중인 스레드가 최대값에 도달하지 않았으면 새로운 스레드를 생성하여 태스크를 실행한다. 최대값에 도달했으면, 0.1초 대기 후 다시 시도하여 스레드 풀에서 자리가 날 때까지 기다린다.

이렇게 c 파일을 작성한 후 헤더파일을 작성해서 다른 파일에서도 사용할 수 있도록 설정해놓는다.

```
// schedule_task.h
#ifndef SCHEDULE_TASK_H
#define SCHEDULE_TASK_H

void scheduleTask(void *(*task)(void *), void *arg);

#endif // SCHEDULE_TASK_H
```

2-2. 소스 수정 과정

i. 터미널 기반 파일 탐색 및 실행 시스템 구현

```
void file_explorer() {
    char command[256], path[256];
    while (1) {
        printf("Enter command (list <directory>, exec <file>, or exit): ");
        if (!fgets(command, sizeof(command), stdin)) break;
        command[strcspn(command, "\n")] = 0;
        if (strcmp(command, "exit") == 0) {
            printf("Exiting file explorer.\n");
            break;
        } else if (sscanf(command, "list %s", path) == 1) {
            list_directory(path);
        } else if (sscanf(command, "exec %s", path) == 1) {
            execute_file(path);
        } else {
            printf("Invalid command. Use 'list <directory>', 'exec <file>', or 'exi
t'.\n");}
    }
}
```

함수 'file_explorer'는 파일 시스템을 탐색하기 위한 간단한 명령줄 인터페이스를 제공합니다. 이 함수는 사용자가 입력한 명령에 따라 디렉터리의 내용을 나열하거나 파일을 실행하거나 프로그램을 종료하며 주요 기능으로는 명령 입력 대기와 명령 처리가 있습니다.

명령 입력 대기에서는 사용자가 명령을 입력할 수 있도록 프롬프트를 표시하고, 입력된 명령을 읽고 개행 문자를 제거합니다. 명령처리에서는 세가지의 명령어를 처리합니다. 'exit' 명령은 프로그램을 종료합니다. 'list <directory>' 명령은 'list_directory' 함수를 호출해서 지정된 디렉터리의 내용을 나열합니다. 'exec <file>' 명령은 'execute_file' 함수를 호출해서 지정된 파일을 실행합니다. 해당 명령어들 이외의 명령어들은 유효하지 않은 명령이라는 메시지를 출력합니다.

이 함수는 루프 안에서 계속 실행되며, 'exit' 명령을 받을 때까지 반복합니다.

```c
void list_directory(const char *path) {
    DIR *dp = opendir(path);
    if (!dp) {
        perror("opendir");
        return;
    }

    struct dirent *entry;
    while ((entry = readdir(dp))) {
        printf("%s\n", entry->d_name);
    }

    closedir(dp);
}
```

```c
void execute_file(const char *path) {
    struct stat sb;
    if (stat(path, &sb) != 0) {
        printf("File does not exist or is not executable\n");
        return;
    }

    pid_t pid = fork();
    if (pid == 0) {
        char *path_copy = strdup(path);
        char *file_name = basename(path_copy);
        char *extension = strrchr(file_name, '.');
        if (extension && (strcmp(extension, ".txt") == 0 || strcmp(extension, ".c") ==
0)) {        execl("/bin/cat", "cat", path, NULL);
        } else {
            execl(path, file_name, NULL);
        }

        perror("execl");
        free(path_copy);
        exit(EXIT_FAILURE);
    } else if (pid < 0) {
        perror("fork");
    } else {
        waitpid(pid, NULL, 0);
    }
}
```

ii. 터미널 파일 탐색 및 정렬 시스템

```
void filesearch(char* path, char* target) {
    char filename[NAMEMAX];
    char pathbuf[PATHMAX];
    char pathbuf2[PATHMAX];

    if (path == NULL && target == NULL) {
        fprintf(stdout, "input filename: ");
        fgets(filename, NAMEMAX, stdin);
        char* temp = strchr(filename, '\n');
        if (temp) *temp = 0;

#ifdef _WIN32
        strcpy(pathbuf, getenv("USERPROFILE"));
        _chdir(getenv("USERPROFILE"));

#else
        strcpy(pathbuf, getenv("HOME"));
        chdir(getenv("HOME"));

#endif
        head = NULL;
        tail = NULL;
    }

    else {
        strcpy(pathbuf, path);
        strcpy(filename, target);
    }

#ifdef _WIN32
    WIN32_FIND_DATA findFileData;
    HANDLE hFind = INVALID_HANDLE_VALUE;

    // Prepare directory search pattern
    char searchPattern[PATHMAX];
    sprintf(searchPattern, "%s/*", pathbuf);
    hFind = FindFirstFile(searchPattern, &findFileData);

    do {
        if (strcmp(findFileData.cFileName, ".") == 0 || strcmp(findFileData.cFileName, "..") =
= 0) {      continue;
        }
        sprintf(pathbuf2, "%s/%s", pathbuf, findFileData.cFileName);
        if (findFileData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) {
            filesearch(pathbuf2, filename);
        }
        else {
            if (strcmp(findFileData.cFileName, filename) == 0) {
                addLink(pathbuf2);
            }
        }
    }
    while (FindNextFile(hFind, &findFileData) != 0);
        FindClose(hFind);
```

```
#else
    struct dirent** list;
    struct stat statbuf;
    int cnt = scandir(pathbuf, &list, NULL, alphasort);

    for (int i = 0; i < cnt; i++) {
        if (strcmp(list[i]->d_name, ".") == 0 || strcmp(list[i]->d_name, "..") == 0) {
            free(list[i]);
            continue;
        }

        sprintf(pathbuf2, "%s/%s", pathbuf, list[i]->d_name);
        if (lstat(pathbuf2, &statbuf) != 0) {
            fprintf(stderr, "lstat err \n");
            exit(1);
        }
        if (S_ISREG(statbuf.st_mode)) {
            if (strcmp(list[i]->d_name, filename) == 0) {
                addLink(pathbuf2);
            }
        }
        else if (S_ISDIR(statbuf.st_mode)) {
            filesearch(pathbuf2, filename);
        }
        free(list[i]);
    }
    free(list);

#endif
    if (path == NULL && target == NULL) {
        printList();
    }
}

#ifdef _WIN32
int main(int argc,char* argv)
{
    filesearch(NULL, NULL);
}

#endif
```

함수 'filesearch'는 지정된 경로에서 특정 파일을 검색하는 기능을 제공하며 주어진 경로가 없으면 사용자의 홈 디렉터리에서 검색을 시작한다. 이 함수는 Windows와 Unix-like 시스템에서 모두 작동하도록 작성되었으며 시스템에 따라 구분되게 작동되도록 작성되었다.

우선 함수가 호출되면 'path'와 'target'이 NULL인지 확인한 후 NULL이면 사용자로부터 검색할 파일 이름을 입력받는다. 입력받은 파일 이름에서 개행 문자를 제거한 후 사용자 홈 디렉토리를 'pathbuf'에 설정하고 파일 구조체인 linked list의 헤드와 테일 포인터를 초기화한다. 만약 'path'와 'target'이 NULL이 아닌 경우에는 'pathbuf'에 주어진 경로를, 'filename'에 주어진 파일 이름을 복사한다. 이

후 디렉토리 내 첫 번째 파일을 찾고 루프를 통해 디렉토리 내 모든 파일과 서브디렉토리를 탐색한다. 현재 디렉토리('.')와 부모 디렉토리 ('..')를 건너뛰면서 서브디렉토리를 찾게 되면, 파일 이름이 'filename'과 일치하는지 확인하고, 일치하면 해당 파일 경로를 linked list에 추가한다. 그리고 다음파일을 찾게 되는데 모든 파일을 탐색한 후에는 검색을 종료한다. 그리고 'path'와 'target'이 NULL인 경우, 검색이 완료된 후 linked list에 저장된 결과를 출력한다.

이 함수는 디렉토리를 재귀적으로 탐색하며, 지정된 파일을 찾으면 그 경로를 linked list에 저장한 후 검색이 완료되면 결과를 출력한다.

```
void addLink(char* path) {

    fileLinkedList* rt = (fileLinkedList*)malloc(sizeof(fileLinkedList));
    strcpy(rt->path, path);
    rt->next = NULL;

    if (head == NULL) {
        head = rt;
        tail = rt;
        rt->prev = NULL;
    }
    else {
        rt->prev = tail;
        tail->next = rt;
        tail = rt;
    }
}
```

```
void printList() {
    fileLinkedList* temp = head;

    if (temp == NULL) {
        fprintf(stdout, "file not founded \n");
    }

    while (temp != NULL) {
        fprintf(stdout, "%s \n", temp->path);
        temp = temp->next;
    }

    // Clear the linked list after printing
    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }
    tail = NULL;
}
```

iii. 터미널 파일 및 폴더 관리 시스템

```
// 폴더 생성 함수
// createFolder 함수
void createFolder() {
    char folderPath[1024];
    printf("생성하고자 하는 폴더 경로를 입력하세요: ");
    fgets(folderPath, sizeof(folderPath), stdin);
    folderPath[strcspn(folderPath, "\n")] = '\0';  // 개행 문자 제거

    char folderName[256];
    printf("생성하고자 하는 폴더 명을 입력하세요: ");
    fgets(folderName, sizeof(folderName), stdin);
    folderName[strcspn(folderName, "\n")] = '\0';  // 개행 문자 제거

    // FolderArgs 구조체에 정보 저장
    FolderArgs *args = (FolderArgs *)malloc(sizeof(FolderArgs));
    args->folderPath = strdup(folderPath);
    args->folderName = strdup(folderName);

    // scheduleTask 함수를 사용하여 스레드 생성 및 createFolderTask 실행
    scheduleTask(createFolderTask, (void *)args);
}
#endif

free(args->folderPath);
free(args->folderName);
free(args);
//printf("thread exit\n");                //debug-log
return NULL;
}
```

'createFolder' 함수는 사용자로부터 폴더 경로와 이름을 입력받고, 이를 'FolderArgs' 구조체에 저장한다. 이를 'scheduleTask' 함수를 통해 새로운 스레드를 생성하여 'creaetFolderTask' 함수를 실행한다. 'createFolderTask' 함수는 입력받은 경로에 새로운 폴더를 생성하고, 결과를 출력한 후 동적으로 할당된 메모리를 해제한다. 이 함수는 멀티스레딩을 사용하여 폴더 생성 작업을 비동기적으로 처리함으로써, 메인 프로그램의 실행을 블로킹하지 않고 폴더 생성 작업을 백그라운드에서 수행한다.

```c
void deleteFolder() {
    char folderPath[1024];
    printf("삭제하고자 하는 폴더 경로를 입력하세요: ");
    fgets(folderPath, sizeof(folderPath), stdin);
    folderPath[strcspn(folderPath, "\n")] = '\0';  // 개행 문자 제거

    char *folderPathCopy = strdup(folderPath);
    scheduleTask(deleteFolderTask, (void *)folderPathCopy);
}
```

```c
// 폴더 삭제 함수
void *deleteFolderTask(void *arg) {
    //printf("thread created\n");                //debug-log
    char *folderPath = (char *)arg;

    #ifdef _WIN32
    if (rmdir(folderPath) == 0) {
        printf("폴더 삭제 성공: %s\n", folderPath);
        printf(">");
    } else {
        perror("폴더 삭제 실패");
        printf(">");
    }
    #else
    if (rmdir(folderPath) == 0) {
        printf("폴더 삭제 성공: %s\n", folderPath);
        printf(">");
    } else {
        perror("폴더 삭제 실패");
        printf(">");
    }
    #endif

    free(folderPath);
    //printf("thread exit\n");                //debug-log
    return NULL;
}
```

'deleteFolder' 함수는 사용자로부터 폴더 경로를 입력받고, 이를 동적으로 할당된 문자열로 복사한다. 이를 'scheduleTask' 함수를 통해 새로운 스레드를 생성하여 'deleteFolderTask' 함수를 실행시킨다. 'deleteFolderTask' 함수는 입력받은 경로의 폴더를 삭제하고, 결과를 출력한 후 동적으로 할당된 메모리를 해제한다. 이 소스도 'createFolder'와 마찬가지로 멀티 스레딩을 사용하여 폴더 삭제 작업을 비동기적으로 처리하고, 이를 통해 메인 프로그램의 실행을 블로킹하지 않고 폴더 삭제 작업을 백그라운드에서 수행한다.

```c
void copyFile() {
    char sourceFileName[1024];
    printf("복사하고자 하는 파일 명을 전체 경로로 입력하세요: ");
    fgets(sourceFileName, sizeof(sourceFileName), stdin);
    sourceFileName[strcspn(sourceFileName, "\n")] = '\0';  // 개행 문자 제거

    char destinationFileName[1024];
    printf("복사하려는 위치를 전체 경로로 입력하세요: ");
    fgets(destinationFileName, sizeof(destinationFileName), stdin);
    destinationFileName[strcspn(destinationFileName, "\n")] = '\0';  // 개행 문자
제거

    char **fileNames = (char **)malloc(2 * sizeof(char *));
    fileNames[0] = strdup(sourceFileName);
    fileNames[1] = strdup(destinationFileName);

    scheduleTask(copyFileTask, (void *)fileNames);
}
```

'copyFile' 함수는 사용자로부터 원본 파일 경로와 대상 파일 경로를 입력받고, 이를 동적으로 할당된 문자열로 복사합니다. 'scheduleTask' 함수를 통해 새로운 스레드를 생성하여 'copyFileTask'함수를 실행하고 'copyFileTask' 함수는 원본 파일을 읽고, 대상 파일에 내용을 복사한 후, 결과를 출력하고 동적으로 할당된 메모리를 해제합니다.

```c
// 파일 복사 함수
void *copyFileTask(void *arg) {
    //printf("thread created\n");                //debug-log
    char **fileNames = (char **)arg;
    char *sourceFileName = fileNames[0];
    char *destinationFileName = fileNames[1];
    FILE *sourceFile, *destinationFile;
    char ch;

    sourceFile = fopen(sourceFileName, "r");
    if (sourceFile == NULL) {
        perror("원본 파일을 열 수 없음");
        free(sourceFileName);
        free(destinationFileName);
        free(fileNames);
        return NULL;
    }

    destinationFile = fopen(destinationFileName, "w");
    if (destinationFile == NULL) {
        perror("대상 파일을 생성할 수 없음");
        fclose(sourceFile);
        free(sourceFileName);
        free(destinationFileName);
        free(fileNames);
        return NULL;
    }

    while ((ch = fgetc(sourceFile)) != EOF) {
        fputc(ch, destinationFile);
    }

    printf("파일 복사 성공: %s -> %s\n", sourceFileName, destinationFileName);

    fclose(sourceFile);
    fclose(destinationFile);

    free(sourceFileName);
    free(destinationFileName);
    free(fileNames);

    //printf("thread exit\n");                //debug-log
    return NULL;
}
```

3. 기능

3-1. 터미널 기반 파일 탐색 및 실행 시스템 구현

주요 기능

i. 폴더 경로 입력 시 하위 항목 나열

사용자가 폴더 경로를 입력하면 해당 폴더 내의 파일 및 디렉토리 목록을 터미널에 출력합니다.

ii. 파일 경로 입력 시 파일 실행

사용자가 파일 경로를 입력하면 해당 파일을 실행합니다.

3-2. 터미널 파일 탐색 및 정렬 시스템

주요 기능

i. 파일 검색 및 정렬

지정된 디렉토리 내에서 파일을 검색하고, 결과를 이름, 날짜, 크기 등 다양한 기준으로 정렬할 수 있습니다.

3-3. 터미널 파일 및 폴더 관리 시스템

주요기능

i. 폴더 생성 및 삭제

사용자가 명령을 통해 새로운 폴더를 생성하거나 기존 폴더를 삭제할 수 있습니다.

ii. 파일 복사 및 붙여넣기

특정 파일을 복사한 후, 원하는 위치에 붙여 넣을 수 있습니다.

4. 결론

기대 효과 및 기존 파일 탐색기와의 차이점

- 성능 향상:
  파일 탐색기는 기존 방법과는 다르게 병렬 처리 및 비동기적 작업을 통해 파일 탐색 작업이 효율적으로 처리되며, 메모리 효율성도 높아 대규모 파일 시스템에서도 우수한 성능을 제공합니다.
- 사용자 경험 개선:
  새로운 파일 탐색기는 사용자의 작업 효율성을 향상시키고 편의성을 높여줍니다. 비동기적 처리를 통해 사용자는 대기 시간을 최소화하고 빠르게 작업을 완료할 수 있으며, 직관적인 사용자 인터페이스는 쉽게 찾고 관리할 수 있도록 도와줍니다.
- 확장성 및 유연성:
  새로운 파일 탐색기는 모듈화된 설계와 플랫폼 독립성을 통해 확장성과 유연성을 제공합니다. 모듈화된 설계는 새로운 기능 추가와 기존 기능 수정을 용이하게 하며, 다양한 운영 체제에서의 호환성은 사용자들이 다양한 환경에서 프로그램을 사용할 수 있도록 합니다.
- 플랫폼 독립성:
  새로운 파일 탐색기는 Windows, Linux, macOS 등 다양한 플랫폼에서 작동합니다. 이는 사용자들이 자신이 선호하는 운영 체제에서 프로그램을 자유롭게 사용할 수 있도록 합니다.